Tuuurbine: A Generic CBR Engine Over RDFS*

Emmanuelle Gaillard, Laura Infante-Blanco,
Jean Lieber, and Emmanuel Nauer

Université de Lorraine, LORIA — 54506 Vandœuvre-lès-Nancy, France
CNRS — 54506 Vandœuvre-lès-Nancy, France
Inria — 54602 Villers-lès-Nancy, France
emmanuelle.gaillard@loria.fr, laura.infanteblanco.@loria.fr,
jean.lieber@loria.fr, emmanuel.nauer@loria.fr

Abstract. This paper presents TUUURBINE, a case-based reasoning (CBR) system for the Semantic Web. TUUURBINE is built as a generic CBR system able to reason on knowledge stored in RDF format; it uses Semantic Web technologies like RDF/RDFS, RDF stores, SPARQL, and optionally Semantic Wikis. TUUURBINE implements a generic case-based inference mechanism in which adaptation consists in retrieving similar cases and in replacing some features of these cases in order to obtain one or more solutions for a given query. The search for similar cases is based on a generalization/specialization method performed by means of generalization costs and adaptation rules. The whole knowledge (cases, domain knowledge, costs, adaptation rules) is stored in an RDF store.
Keywords: generic CBR engine, Semantic Web, RDFS, triple store

1 Introduction and motivations

This paper presents TUUURBINE (`http://tuuurbine.loria.fr/`), a new generic case-based reasoning system (CBR), the reasoning procedure of which is based on a domain ontology. TUUURBINE was created on the basis of the experience acquired during five years with the TAAABLE system [1].

This research work is motivated by the will to develop a generic CBR system. TUUURBINE implements a generic case-based inference. Searching for similar cases is based on a generalization/specialization process performed by means of generalization costs and adaptation rules. The generic adaptation approach is inspired from the TAAABLE's adaptation.

The second major motivation is the development of a system able to exploit the huge and growing amount of knowledge available on the Web, especially the knowledge coming from The Linked Data or contained in Semantic Wikis.

* The development of TUUURBINE was supported by an Inria ADT funding from October 2011 to October 2013. The authors would like to thank the reviewers who have helped improving the quality of this paper: it was not possible to take into account all their remarks in the paper, but these remaining remarks point out interesting issues that the authors plan to address as future work.

This is why Tuuurbine is built according to Semantic Web standards (RDF, SPARQL, RDFS), to facilitate the interoperability with Semantic Web knowledge. With Tuuurbine, the knowledge (cases, domain knowledge, costs, adaptation rules) is encoded in a triple store, which may be installed in the same machine as the reasoner or in a remote one, and could be additionally be interfaced with a Semantic Wiki, in order to benefit from collaborative web edition and knowledge management involved in the reasoning process [2].

Storing the knowledge in a triple store provides an important advantage: the knowledge is not managed by the CBR system anymore, but by an external tool that is efficient and based on standards. A clear separation is made between the reasoning inference engine sub-system and the knowledge handling and storage sub-system of the CBR scheme: the knowledge used by the CBR system is not loaded in cache once and for all, but on demand, only the necessary knowledge is retrieved to solve a new problem, using SPARQL language. This approach also ensures the exploitation of up-to-date knowledge to best solve a problem in environments where the knowledge evolves continuously. The choice of such an architecture is the result of the experience acquired with Taaable. Indeed, in the first version of Taaable, the knowledge was completely loaded in memory from several files encoded in various languages [3]. This caused the problem of impersistence of updated knowledge cache, which appears frequently when tuning the system. The second version of Taaable improved the knowledge evolution using a Semantic Wiki, called WikiTaaable [2] enabling collaborative edition of the knowledge, but still the Taaable CBR system required, at that time, loading a new dump of knowledge in memory, after every single knowledge modification, in order to use up-to-date knowledge. The last version of Taaable, which uses now the Tuuurbine CBR engine, benefits from a complete synchronization with the knowledge base thanks to a dynamic access to the triple store.

The paper is organized as follows. Section 2 describes briefly the Semantic Web technologies used in Tuuurbine. Section 3 describes the generic CBR approach of Tuuurbine and details the knowledge the approach is based on and the knowledge representation choices that were made. Section 4 details the architecture of the system. Section 5 presents two use-cases. Section 6 discusses this work and relates it to other works. Section 7 concludes the paper.

2 Semantic Web technologies used in Tuuurbine

RDF, RDFS and the Triple Stores. RDF[1], SPARQL[2] and RDFS[3] are three standards recommendations of the W3C (World Wide Web Consortium) for the Semantic Web.

RDF (Resource Description Framework) is a format to encode resources over the Web, existing in various syntaxes. A *resource* is any kind of entity that has been reified (i.e., associated with an identifier). A *literal* is a constant datatype

[1] `http://www.w3.org/RDF`
[2] `http://www.w3.org/2009/sparql`
[3] `http://www.w3.org/TR/rdf-schema`

(e.g., an integer, a float, a string, etc.). A *property* relates a resource to another resource or a literal. An RDF base is a set of *triples*, a triple being an expression of the form $\langle s \;\; p \;\; o \rangle$. s, p and o are respectively called the *subject* (a resource), the *predicate* (a property) and the *object* (a resource or a literal) of a triple. For example, if `romeo` and `juliet` are two resources to be understood as the main characters of [4], if `loves` and `ofTheFamily` are the properties meaning "loves" and "is a member of the family", and if `age` relates a resource to a literal of integer datatype indicating the age of the resource, then the RDF base $\mathcal{B} = \{\langle \texttt{romeo ofTheFamily montague} \rangle, \langle \texttt{romeo loves juliet} \rangle, \langle \texttt{juliet age 13} \rangle\}$ means that Romeo, a member of the family Montague, loves Juliet who is 13.

SPARQL (recursive acronym for SPARQL Protocol And RDF Query Language) is at the same time a protocol and a query language for RDF. It can be likened to SQL: SPARQL is to RDF as SQL is to relational databases. For instance, the following SPARQL query to the base $\mathcal{B}$ above returns the resources x such that $\langle x \;\; \texttt{loves juliet} \rangle \in \mathcal{B}$ and $\langle x \;\; \texttt{ofTheFamily montague} \rangle \in \mathcal{B}$:

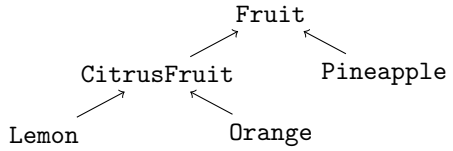
```
SELECT ?x
WHERE { ?x loves juliet .                    // "." stands for "and"
        ?x ofTheFamily montague}
```

RDFS (RDF Schema) can be considered as a knowledge representation formalism, the syntax of which is RDF (a formula of RDFS is an RDF triple) and the semantics of which is associated with a set of resources having a predefined semantics, called the RDFS *vocabulary*. In this paper, only the properties `rdf:type` and `rdfs:subClassOf` of the RDFS vocabulary are considered and they are abbreviated by `type` and `subc`. `type` can be understood as "is an element of" ($\in$) and `subc` as "is a subset of" ($\subseteq$). The following inference rules are used for RDFS entailment reduced to the vocabulary $\{\texttt{type}, \texttt{subc}\}$:

$$\frac{\langle a \;\; \texttt{type} \;\; C \rangle \qquad \langle C \;\; \texttt{subc} \;\; D \rangle}{\langle a \;\; \texttt{type} \;\; D \rangle} \qquad \frac{\langle C \;\; \texttt{subc} \;\; D \rangle \qquad \langle D \;\; \texttt{subc} \;\; E \rangle}{\langle C \;\; \texttt{subc} \;\; E \rangle}$$

`subc` enables to define hierarchies of classes based on the "is more specific than" relation. Moreover, if C denotes an RDFS class, then $\langle C \;\; \texttt{subc} \;\; C \rangle$ is an RDFS axiom. If $\mathcal{B}$ is an RDFS base and τ is a triple, $\mathcal{B} \vdash \tau$ means that τ is a logical consequence of $\mathcal{B}$. Thus, $\mathcal{B} \nvdash \tau$ means that τ cannot be entailed from $\mathcal{B}$.

For example, the hierarchy



can be represented by the following RDFS base:

$$\mathcal{H} = \{\langle \texttt{CitrusFruit subc Fruit} \rangle, \;\; \langle \texttt{Pineapple subc Fruit} \rangle, \\ \langle \texttt{Lemon subc CitrusFruit} \rangle, \;\; \langle \texttt{Orange subc CitrusFruit} \rangle\}$$

which entails, e.g., the triple $\langle \texttt{Orange subc Fruit} \rangle$, i.e. every orange is a fruit.

Let SQ be a SPARQL query and $\mathcal{B}$ be a RDFS base. Then $\texttt{Result}_{\vdash}(\texttt{SQ}, \mathcal{B})$ denotes (in this paper) the result of the SPARQL query on $\mathcal{B}$ using the entailment. For example,

$$\texttt{Result}_{\vdash}\left(\boxed{\begin{array}{l}\texttt{SELECT ?x}\\ \texttt{WHERE \{?x subc CitrusFruit\}}\end{array}}, \mathcal{H}\right) = \left\{\begin{array}{l}\texttt{Lemon, Orange,}\\ \texttt{CitrusFruit}\end{array}\right\}.$$

A *triple store* (also called RDF *store*) engine is a database management system for RDF. Using an RDF store enables in particular to avoid loading the whole RDF knowledge base in cache. Some triple store engines support RDFS entailment. However, for performance reasons, in the current version of TUUURBINE, the triple store chosen by default is 4Store[4] which does not draw entailments, these being implemented by TUUURBINE.

Semantic Wikis. A *semantic wiki* is a wiki the contents of which are not only documents and links between documents (as in classical Wikis) but also machine-processible data. SMW is a Semantic Wiki engine extending MediaWiki.[5] The semantic data in a semantic wiki using SMW are managed via a triple store. From a knowledge engineering viewpoint, a semantic wiki engine can be considered as a cooperative knowledge management tool: the knowledge can be edited thanks to this tool, associated with (non formalized) pieces of knowledge in plain text. To exploit the data of a Semantic Wiki, one can create a dump to an RDF file or query directly the triple store, using a SPARQL endpoint.

3 Tuuurbine reasoning principles

This section presents the principles upon which the TUUURBINE engine has been implemented. First, the running example is introduced. This example is then developed in the subsequent sections: representation of pieces of knowledge (cases, domain knowledge, similarity and adaptation knowledge), the representation of TUUURBINE queries, and the retrieval and adaptation procedures.

3.1 Introduction of the running example

Let us consider an application of CBR where a case is a recipe (and a case base represents a recipe book). Such application has been developed for the Computer Cooking Contest at ICCBR for the past years.

In this application, let us consider the following query:

$$\texttt{Q} = \boxed{\text{a cocktail recipe with mint, gin, orange juice but no wine.}} \qquad (1)$$

If at least one recipe matches exactly Q, the application returns it. Otherwise, a recipe matching approximately Q is searched (retrieval step) and then is modified in order to answer Q (adaptation step).

[4] http://4store.org
[5] http://semantic-mediawiki.org/

Let us assume that no recipe matching exactly `Q` can be found but that the following source case similar to `Q` is retrieved:

`Source` =

Recipe "Mexican cocktail in my way"
Dish type: cocktail
Ingredients: 40 cl tequila, 1 l guava juice, 1 l pineapple juice, 30 cl apple juice, 1 pkt vanilla sugar, 3 mint leaves
Preparation: Combine guava juice and pineapple. Add tequila and apple juice. [...]

`Source` matches approximately `Q` since:

- There is an exact match on the dish type (cocktail), the ingredient mint and the absence of the ingredient wine.
- The ingredients `Tequila` (`Source`) and `Gin` (`Q`) are subclasses of `Liquor`, and the ingredients `GuavaJuice`, `PineappleJuice`, `AppleJuice` (`Source`) and `OrangeJuice` (`Q`) are subclasses of `FruitJuice`.

Finally, adaptation modifies `Source` so that the modified recipe matches exactly `Q`. The adaptation is usually based on the approximate matching between `Source` and `Q`. In this example, it consists in applying the following modification:

Replace tequila with gin and guava juice, pineapple juice, or apple juice with orange juice.

(2)

Furthermore, there could be available rules which can be applied to adapt the case. In the example, let us consider the following adaptation rule:

$\texttt{AR}_1$ =

In the context of a cocktail dish without anise, guava juice and vanilla sugar can be substituted with orange juice and sugar cane syrup.

This adaptation rule enables the retrieval of another (hopefully, better) adaptation of `Source`, given by the following modification:

Replace tequila with gin, and guava juice and vanilla sugar with orange juice and sugar cane syrup.

(3)

3.2 Knowledge containers of a Tuuurbine application

A CBR knowledge base can be split into four knowledge containers [5]. This knowledge is represented by an RDFS base `KB` and managed by a triple store.

Representation of cases. In the running example, the reasoning takes into account only the dish type and the ingredients of the recipe (neither its title nor its preparation). Moreover, the ingredient quantities are not taken into account.

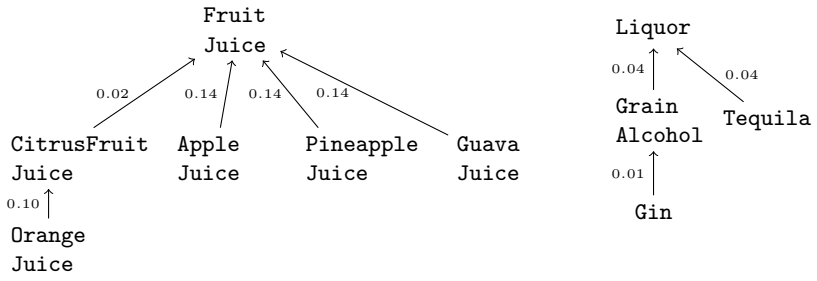


Fig. 1. The hierarchy forming the domain knowledge used in the running example with the generalization costs as retrieval knowledge.

Therefore, this recipe can be represented by the following RDFS base:

$$\texttt{Source} = \left\{ \begin{array}{l} \langle s \ \texttt{type SourceCase} \rangle, \langle s \ \texttt{dishType CocktailDish} \rangle, \\ \langle s \ \texttt{ingredient Tequila} \rangle, \langle s \ \texttt{ingredient GuavaJuice} \rangle, \\ \langle s \ \texttt{ingredient PineappleJuice} \rangle, \langle s \ \texttt{ingredient AppleJuice} \rangle, \\ \langle s \ \texttt{ingredient VanillaSugar} \rangle, \langle s \ \texttt{ingredient Mint} \rangle \end{array} \right\}$$

More generally, a case of the case base is identified by a resource s and is defined by an RDFS base $\texttt{Source} \subseteq \texttt{KB}$ containing the triples of the form $\langle s \ \texttt{prop val} \rangle$: this is a simple feature-value representation (the attribute being `prop` and the value being `val`). The triple $\langle s \ \texttt{type SourceCase} \rangle$ is mandatory to indicate that this case belongs to `CaseBase`, the case base: it enables the obtention of the whole case base via the following SPARQL query:

```
SELECT ?s
WHERE {?s type SourceCase}
```

.

Representation of domain knowledge. The domain knowledge is represented by an RDFS base `DK` ($\texttt{DK} \subseteq \texttt{KB}$) consisting in a set of triples in the form $\langle C \ \texttt{subc} \ D \rangle$. Fig. 1 represents the domain knowledge for the running example by a hierarchy the edges $C \xrightarrow{x} D$ of which represent the triples $\langle C \ \texttt{subc} \ D \rangle$ (the meaning of x is explained hereafter).

Representation of similarity (retrieval knowledge). The retrieval knowledge is encoded by a cost function, associating a triple $\langle C \ \texttt{subc} \ D \rangle \in \texttt{DK}$ to a positive real number $\texttt{cost}(\langle C \ \texttt{subc} \ D \rangle)$ for a given property. If $C \xrightarrow{x} D$ is an edge of the Fig. 1 hierarchy then $\texttt{cost}(\langle C \ \texttt{subc} \ D \rangle) = x$. This cost can be understood intuitively as the measure of "the generalization effort" from C to D. The cost function is assumed to be additive:

$$\texttt{cost}(\langle C \ \texttt{subc} \ E \rangle) = \texttt{cost}(\langle C \ \texttt{subc} \ D \rangle) + \texttt{cost}(\langle D \ \texttt{subc} \ E \rangle)$$

Therefore $\texttt{cost}(\langle \texttt{OrangeJuice subc FruitJuice} \rangle) = 0.10 + 0.02 = 0.12$ can be deduced from Fig. 1 and, for any RDFS class C, $\texttt{cost}(\langle C \ \texttt{subc} \ C \rangle) = 0$.

Since it is a tedious work to set manually all these costs, some default values are computed by TUUURBINE according to the following formula:

$$\texttt{cost}(\langle C \ \ \texttt{subc} \ \ D\rangle) = K \cdot \frac{\texttt{\#CasesWith}(D) - \texttt{\#CasesWith}(C)}{|\texttt{CaseBase}|}$$

$$\text{where } \texttt{\#CasesWith}(X) = \left|\texttt{Result}_{\vdash}\left(\boxed{\begin{array}{l}\texttt{SELECT ?x}\\ \texttt{WHERE \{ ?x type SourceCase .}\\ \qquad\qquad \texttt{?x ?p } X\texttt{\}}\end{array}}, \texttt{KB}\right)\right|,$$

K is a coefficient cost factor depending of `?p`

and $|A|$ is the cardinal of the set A

A similar formula is used by the TAAABLE system [1].

Representation of adaptation knowledge. The adaptation of a case `Source` to answer a query `Q` is performed using domain knowledge, retrieval knowledge and, when available, adaptation knowledge in the form of a finite set `AK` of adaptation rules. Syntactically, an adaptation rule is defined as follows:

$$\texttt{AR} = \boxed{\begin{array}{rl}\textbf{p-context}\ p_1, p_2, \ldots & \textit{// positive context}\\ \textbf{n-context}\ n_1, n_2, \ldots & \textit{// negative context}\\ \textbf{replace}\ r_1, r_2, \ldots & \textit{// left part}\\ \textbf{with}\ w_1, w_2, \ldots & \textit{// right part}\end{array}}$$

where p_1, p_2, …, n_1, n_2, …, r_1, r_2, …, w_1, w_2, … are expressions in the form $\texttt{prop}:C$, `prop` being an RDFS property and C being an RDFS class. For example, the adaptation rule introduced in the running example (§3.1) can be formalized by

$$\texttt{AR}_1 = \boxed{\begin{array}{l}\textbf{p-context}\ \texttt{dishType}:\texttt{CocktailDish}\\ \textbf{n-context}\ \texttt{ingredient}:\texttt{Anise}\\ \quad\textbf{replace}\ \texttt{ingredient}:\texttt{GuavaJuice}, \texttt{ingredient}:\texttt{VanillaSugar}\\ \qquad\textbf{with}\ \texttt{ingredient}:\texttt{OrangeJuice}, \texttt{ingredient}:\texttt{SugarCaneSyrup}\end{array}}$$

Let `Source` be a case identified by s. `AR` is applicable on `Source` if:

- For each term $t = \texttt{prop}:C \in \{p_1, p_2, \ldots, r_1, r_2, \ldots\}$, $\texttt{KB} \vdash \langle s \ \ \texttt{prop} \ \ C\rangle$ (recall that $\texttt{Source} \subseteq \texttt{KB}$);
- For each term $t = \texttt{prop}:C \in \{n_1, n_2, \ldots\}$, $\texttt{KB} \nvdash \langle s \ \ \texttt{prop} \ \ C\rangle$.[6]

If `AR` is applicable on `Source` then the application of `AR` on `Source` gives a case `AR`(`Source`) obtained by:

[6] This amounts to a closed world assumption (CWA) defined by the inference rule $\frac{\texttt{KB} \nvdash \langle s \ \ \texttt{prop} \ \ C\rangle}{\neg\langle s \ \ \texttt{prop} \ \ C\rangle}$. CWA is justified by the fact that there is no negation in RDFS. From a CBR viewpoint, this means that a case represents a specific situation (it is not a generalized case [6]): every fact τ that is expressible in the `KB` vocabulary that is not entailed by `Source` is assumed not to hold for `Source`.

- Deleting all the triples of `Source` matching r_1, r_2, ...;
- Adding all the triples $\langle s \ \texttt{prop} \ C \rangle$ for each $\texttt{prop} : C \in \{w_1, w_2, \ldots\}$;
- Substituting all the occurrences of s by s' (a new case identifier).

For example, the rule $\texttt{AR}_1$ is applicable on the recipe `Source` of the running example and $\texttt{AR}_1(\texttt{Source}) = \texttt{AS}$ (adapted `Source`) with:

$$\texttt{AS} = \left\{ \begin{array}{l} \langle s' \ \texttt{type SourceCase} \rangle, \langle s' \ \texttt{dishType CocktailDish} \rangle, \\ \langle s' \ \texttt{ingredient Tequila} \rangle, \langle s' \ \texttt{ingredient PineappleJuice} \rangle, \\ \langle s' \ \texttt{ingredient AppleJuice} \rangle, \langle s' \ \texttt{ingredient Mint} \rangle, \\ \langle s' \ \texttt{ingredient OrangeJuice} \rangle, \langle s' \ \texttt{ingredient SugarCaneSyrup} \rangle \end{array} \right\}$$

In other words, the application of $\texttt{AR}_1$ consists in applying the following substitution to the ingredients:

$$\Sigma = \texttt{GuavaJuice} \wedge \texttt{VanillaSugar} \rightsquigarrow \texttt{OrangeJuice} \wedge \texttt{SugarCaneSyrup}$$

Any adaptation rule $\texttt{AR} \in \texttt{AK}$ has an associated value $\texttt{cost}(\texttt{AR}) > 0$, used during the adaptation process (see Section 3.5).

Finally, TUUURBINE proposes another kind of adaptation rules called "specific adaptation rules" (`SAR`) with $\texttt{cost}(\texttt{SAR}) = 0$. Such a rule is associated to a source case which constitutes its context. For the recipe application, this rule can be seen as a way of encoding variants of the recipe. For example, the piece of information "Basil can be used instead of mint in the recipe of the running example" could be represented by the following specific adaptation rule:

$\texttt{SAR}_1 =$

p-context "Mexican cocktail in my way" recipe
replace `ingredient:Mint`
with `ingredient:Basil`

3.3 Representation of Tuuurbine queries

Syntactically, a TUUURBINE query is a conjunction of expressions in the form `sign prop:val` where $\texttt{sign} \in \{\epsilon, +, !, -\}$, `prop` is an RDF property and `val` is either a resource representing a class or a literal. For example, the following query is a TUUURBINE translation of the query (1):[7]

$$\begin{array}{l} \texttt{Q} = +\texttt{dishType:CocktailDish} \wedge \texttt{ingredient:Mint} \\ \quad \wedge \texttt{ingredient:Gin} \wedge \texttt{ingredient:OrangeJuice} \wedge !\texttt{ingredient:Wine} \end{array} \tag{4}$$

The signs ϵ and $+$ are "positive signs": they prefix features that the requested case must have. $+$ indicates that this feature must also occur in the source case whereas ϵ indicates that the source case may not have this feature, thus the adaptation phase has to make it appear in the final case.

[7] ϵ denotes the empty word. Thus, ϵ`prop:val` is simply written `prop:val`. The $+$ in front of `dishType:CocktailDish` means that the part of the case base searched corresponds to cocktail recipes.

The signs ! and − are "negative signs": they prefix features that the requested case must not have. − indicates that this feature must not occur in the source case whereas ! indicates that the source case may have this feature, and that the adaptation phase has to remove it.

3.4 Case retrieval in Tuuurbine

Let $\mathtt{Q}$ be a TUUURBINE query. The goal of retrieval is to find the cases $\mathtt{Source} \in \mathtt{CaseBase}$ that best match $\mathtt{Q}$. If no source case exactly match $\mathtt{Q}$, then the query is relaxed and an approximate matching is searched.

Exact matching search. $\mathtt{Q}$ can be written $\mathtt{Q} = \bigwedge_i \mathtt{sign}_i\mathtt{prop}_i : \mathtt{val}_i$. For each i, let us consider the SPARQL query $\mathtt{SQ}_i =$

```
SELECT ?s
WHERE { ?s type SourceCase .
        ?s prop_i ?x .
        ?x subc val_i}
```

.

For example, the element of query $+\mathtt{dishType} : \mathtt{CocktailDish}$ gives a SPARQL query that can be read "get the source cases with a cocktail dish type" (i.e., the `?s` such that `?s` is a instance of `SourceCase` and such that `?s` has a dish type `?x` which is a subclass of `CocktailDish`).

The exact matching of the query $\mathtt{Q}$ can be done by executing the SPARQL queries $\mathtt{SQ}_i$ and combining them as follows (`EMS` stands for "Exact Matching Search"):

$$\mathtt{EMS}(\mathtt{Q}) = \bigcap_{i,\mathtt{sign}_i \in \{\epsilon,+\}} \mathtt{Result}_{\vdash}(\mathtt{SQ}_i) \quad \setminus \quad \bigcup_{i,\mathtt{sign}_i \in \{!,-\}} \mathtt{Result}_{\vdash}(\mathtt{SQ}_i)$$

In other words, the result of the exact matching search are the source cases that match the SPARQL queries $\mathtt{SQ}_i$ such that $\mathtt{sign}_i$ is a positive sign and that does not match the SPARQL queries $\mathtt{SQ}_i$ such that $\mathtt{sign}_i$ is a negative sign.[8]

Approximate search. The principle of this search is to find a generalization function Γ with minimal cost such that the execution of the query $\mathtt{Q}$ modified by Γ returns at least one source case: $\mathtt{EMS}(\Gamma(\mathtt{Q})) \neq \emptyset$.

Let $\mathtt{Q} = \bigwedge_i \mathtt{sign}_i\mathtt{prop}_i : \mathtt{val}_i$ be a query. A one step-generalization $\gamma(\mathtt{Q})$ of $\mathtt{Q}$ consists in generalizing a term $\mathtt{sign}_i\mathtt{prop}_i : \mathtt{val}_i$ such that $\mathtt{sign}_i \notin \{+, -\}$:

- If $\mathtt{sign}_i = \epsilon$ and $\mathtt{val}_i$ is an RDFS class, then the generalizations of this term are the terms $\mathtt{sign}_i\mathtt{prop}_i : \mathtt{val}$ such that $\langle \mathtt{val}_i \ \ \mathtt{subc} \ \ \mathtt{val} \rangle \in \mathtt{DK}$. This one step generalization is written $\gamma = \mathtt{prop}_i : \mathtt{val}_i \rightsquigarrow \mathtt{prop}_i : \mathtt{val}$ or, simply, $\gamma = \mathtt{val}_i \rightsquigarrow \mathtt{val}$. The cost of such a generalization is $\mathtt{cost}(\langle \mathtt{val}_i \ \ \mathtt{subc} \ \ \mathtt{val} \rangle)$.
- If $\mathtt{sign}_i = !$ or if $\mathtt{sign}_i = \epsilon$ and $\mathtt{val}_i$ is a literal, then $\mathtt{val}_i$ is directly generalized to the ontology top $\top$. This one step generalization is written $\gamma = \mathtt{val}_i \rightsquigarrow \top$. The cost of such a generalization is 0.

[8] In practice, $\mathtt{EMS}(\mathtt{Q})$ could be computed thanks to the execution of fewer SPARQL queries thus giving the same result with a lower computational cost.

A generalization function Γ is a composition of one-step generalizations γ_1, γ_2, $\ldots \gamma_n$: $\Gamma = \gamma_n \circ \ldots \circ \gamma_2 \circ \gamma_1$. Its cost is the sum of the costs of γ_i. For example, the generalization function Γ can be applied on $\mathtt{Q}$ defined by equation (4):

$$\begin{aligned} \Gamma &= \mathtt{Gin} \rightsquigarrow \mathtt{Liquor} \circ \mathtt{OrangeJuice} \rightsquigarrow \mathtt{FruitJuice} \\ \text{and } \Gamma(\mathtt{Q}) &= +\mathtt{dishType} : \mathtt{CocktailDish} \wedge \mathtt{ingredient} : \mathtt{Mint} \\ &\quad \wedge \mathtt{ingredient} : \mathtt{Liquor} \wedge \mathtt{ingredient} : \mathtt{FruitJuice} \\ &\quad \wedge !\mathtt{ingredient} : \mathtt{Wine} \qquad (5) \\ \mathtt{cost}(\Gamma) &= \mathtt{cost}(\langle \mathtt{Gin\ \ subc\ \ Liquor} \rangle) \\ &\quad + \mathtt{cost}(\langle \mathtt{OrangeJuice\ \ subc\ \ FruitJuice} \rangle) \end{aligned}$$

The execution of the query $\Gamma(\mathtt{Q})$ returns the recipe of the example: $\mathtt{Source} \in \mathtt{EMS}(\Gamma(\mathtt{Q}))$ so, provided that $\mathtt{cost}(\Gamma)$ is the minimum of the costs of the generalization functions Λ such that $\mathtt{EMS}(\Lambda(\mathtt{Q})) \neq \emptyset$, $\mathtt{Source}$ is a retrieved case. Technically, Γ is searched by increasing cost in a generalization function space.

Searching for less similar cases. It may occur that a user of a TUUURBINE application wants to find other cases than the ones returned in a first launch. TUUURBINE offers the possibility to do so: it simply consists in resuming the search after Γ has been found. This way, a second generalization function Γ' can be found. For example:

$$\Gamma' = \mathtt{Gin} \rightsquigarrow \mathtt{Alcohol} \circ \mathtt{Mint} \rightsquigarrow \mathtt{Herb}$$

3.5 Case adaptation in Tuuurbine

There are two adaptation processes in TUUURBINE.

The first adaptation process consists in obtaining the matching between $\mathtt{Source}$ and $\mathtt{Q}$ that has permitted the retrieval of $\mathtt{Source}$, this matching being composed of the matching between $\mathtt{Source}$ and $\Gamma(\mathtt{Q})$ (given by the fact that $\mathtt{Source} \in \mathtt{EMS}(\Gamma(\mathtt{Q}))$) and the matching between $\Gamma(\mathtt{Q})$ and $\mathtt{Q}$ (given by the generalization function Γ). This first kind of adaptation, for the running example, works as follows. $\Gamma(\mathtt{Q})$ defined by equation (5) matches exactly $\mathtt{Source}$ (wrt to $\mathtt{DK}$). In fact, there are several matchings between $\mathtt{Source}$ and $\Gamma(\mathtt{Q})$:

$\mathtt{Tequila}$ matches $\mathtt{Liquor}$

each $ing \in \{\mathtt{GuavaJuice}, \mathtt{PineappleJuice}, \mathtt{AppleJuice}\}$ matches $\mathtt{FruitJuice}$

By composing this matching between $\mathtt{Source}$ and $\Gamma(\mathtt{Q})$ and the matching between $\Gamma(\mathtt{Q})$ and $\mathtt{Q}$ given by Γ, it comes the following adaptation:

In the "Mexican cocktail in my way" recipe, substitute $\mathtt{Tequila}$ by $\mathtt{Gin}$

and substitute $\left|\begin{array}{l}\mathtt{GuavaJuice} \text{ and/or} \\ \mathtt{PineappleJuice} \text{ and/or} \\ \mathtt{AppleJuice}\end{array}\right|$ by $\mathtt{OrangeJuice}$.

which represents the expected adaptation (2).

The second kind of adaptation uses the adaptation rules $\texttt{AR} \in \texttt{AK}$ *and* the query generalization procedure used for case retrieval. More precisely it searches a pair (Σ, Γ) such that:

- $\Sigma(\texttt{Source})$ matches exactly $\Gamma(\texttt{Q})$ (wrt DK);
- Σ is a composition of adaptation rules that is applicable on Source ($\Sigma = \texttt{AR}_p \circ \ldots \circ \texttt{AR}_2 \circ \texttt{AR}_1$ such that $\texttt{AR}_1$ is applicable on Source, $\texttt{AR}_2$ is applicable on $\texttt{AR}_1(\texttt{Source})$, etc.);
- Γ is a generalization function;
- $\texttt{cost}(\Sigma)+\texttt{cost}(\Gamma)$ is minimal (where $\texttt{cost}(\Sigma) = \texttt{cost}(\texttt{AR}_p)+\ldots+\texttt{cost}(\texttt{AR}_2)+\texttt{cost}(\texttt{AR}_1)$).

Technically, this kind of adaptation relies on a best-first search: the states are pairs (Σ, Γ); a final state is such that $\Sigma(\texttt{Source})$ matches exactly $\Gamma(\texttt{Q})$; the state space is searched by increasing $\texttt{cost}(\Sigma) + \texttt{cost}(\Gamma)$. It must be noticed that:

- $\texttt{cost}(\Sigma) + \texttt{cost}(\Gamma) \leq \texttt{cost}(\Gamma_{\text{retrieval}})$ where $\Gamma_{\text{retrieval}}$ is the generalization function generated during retrieval (and used by the first kind of adaptation);
- If $\texttt{AK} = \emptyset$ then Σ is the identity function and $\Gamma = \Gamma_{\text{retrieval}}$.

For the example, if $\texttt{AK} = \{\texttt{AR}_1\}$ and $\texttt{cost}(\texttt{AR}_1) + \texttt{cost}(\langle\texttt{Gin subc Liquor}\rangle) < \texttt{cost}(\Gamma_{\text{retrieval}})$ then the adaptation coincides with (3).

4 Implementation

The general architecture of TUUURBINE is presented in Fig. 2. Its CBR engine is composed of three main modules: the RDF store manager, the case manager and the reasoning manager. The case manager implements EMS (the exact matching search).The case retrieval triggers the RDF store manager, in charge of generating and executing a set of SPARQL queries, using a SPARQL endpoint. The results are returned to the case manager which combines them to build the cases.

When no case satisfies the EMS, the reasoning manager is triggered for an approximative search. A first step consists in loading the generalization costs from the triple store (only the costs that are potentially useful). This is done once, at the beginning of the approximative search process. The generalisation process iterates until the EMS returns at least one case for a generalized query. These cases are then adapted using possibly adaptation knowledge and generalization knowledge as explained in §3.5.

TUUURBINE is implemented as a web service. This allows to use HTTP requests to query the CBR engine, but a generic user interface is also provided (an instantiation of this GUI is visible in Fig. 3). Additionally, a Semantic Wiki may be used to manage the whole knowledge in a more convenient way, as it is done for example in WIKITAAABLE, a semantic wiki in the domain of cooking.

TUUURBINE required about 6000 lines of Java code. The user interface is developed in PHP, javascript, HTML/CSS. The exchange between the interface

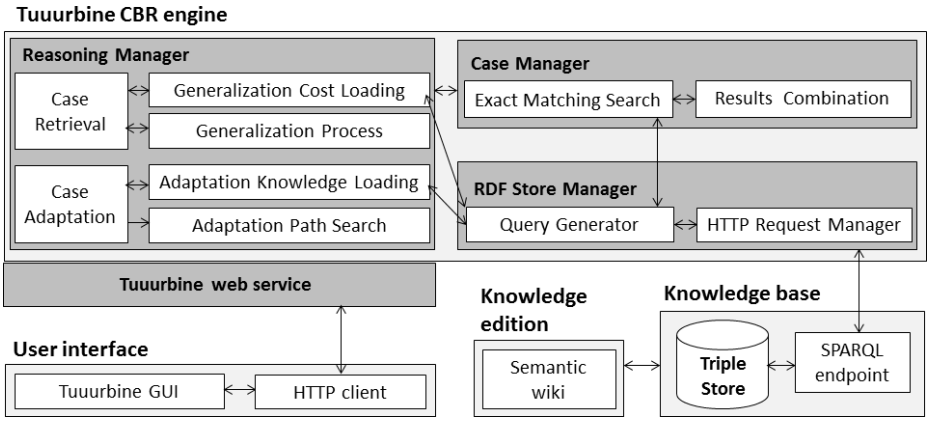


Fig. 2. Architecture of TUUURBINE

and the CBR engine, as well as the configuration files are based on JSON encoding language. TUUURBINE is distributed under an Affero GPL Licence and is available from `http://tuuurbine.loria.fr/`.

The retrieval time for the query (4) is 3156 ms on a server Intel Xeon E5520 64 bits with 8×2.27 GHz processors, and a 32 GB RAM, running on Linux Ubuntu 12.4. 85% of this time is for disk access and data communication. It required the execution of 117 SPARQL queries. This example is realistic: the RDF base size is 277 megabytes, it contains more than 10^6 triples, 1641 of them being triples of the form $\langle s \;\; \texttt{type Recipe} \rangle$ (hence 1641 source cases) and 2316 triples of the form $\langle C \;\; \texttt{subc} \;\; D \rangle$ constitute the domain knowledge. A systematic performance study of TUUURBINE remains to be done as future work.

5 Tuuurbine in action

The first use case is the instantiation of TUUURBINE in the cooking domain, as the new version of TAAABLE, a CBR system which retrieves and creates cooking recipes by adaptation that has been developed to participate in the Computer Cooking Contest[9] since 2008 [3]. Fig. 3 presents the TUUURBINE interface running query 4 and is reachable at `http://tuuurbine.loria.fr/taaable/`. The TAAABLE knowledge base (`http://wikitaaable.loria.fr/`) is composed of the four classical knowledge containers: (1) the domain knowledge as an ontology of the cooking domain which includes several hierarchies (about food, dish types, etc.), (2) the case base, which are recipes described by their titles, the dish type they produces, the ingredients which are required, the preparation steps, etc., (3) the adaptation knowledge takes the form of adaptation rules as

[9] `http://computercookingcontest.net`

Fig. 3. Generic TUUURBINE interface applied to the cooking domain.

introduced in this paper, and (4) the retrieval knowledge which is stored as cost values on subclass-of relations and adaptation rules.

To show that TUUURBINE is independent of the application domain, a second instantiation of TUUURBINE has been implemented in the domain of music. This application use case is about searching for a piece of music you may play using a given set of instruments. The cases are pieces of music described by its composer(s), its music genre(s), origin(s) and year of creation, and the instruments required for playing them. The adaptation consists in replacing some instruments by other ones. The music ontology and the TUUURBINE interface for the music CBR system can be respectively reached at `http://muuusic.loria.fr/` and `http://tuuurbine.loria.fr/muuusic/`.

6 Discussion and related work

Several tools address CBR in a generic way. myCBR and jCOLIBRI are probably the most famous of them. jCOLIBRI [7] is an object-oriented framework for developing CBR applications. This framework includes connectors with several kinds of data sources (database, XML, etc.) for loading the cases in cache, on which the retrieval, reuse, revise and retain tasks can be performed. Specialized modules take into account various types of CBR applications, like CBR on textual cases, data or knowledge-intensive CBR. With jCOLIBRI, the knowledge intensive CBR approach consists in retrieving cases according to an ontology based similarity measure and in replacing some case features (instance of an ontology concept) with the closest ones in the ontology (based on similarity minimization). TUUURBINE also follows this substitution approach but the way the retrieval and the adaptation of cases are made is very different. Indeed, using

the structure of the ontology only in order to compute a numerical measure has a major limitation, due to the lack of semantics of the measure, which can, moreover, be computed with various similarity functions (cosine, fdeep, etc.) which take into account various criteria (depth of the two concepts c_1 and c_2 which are compared, depth and relations of c_1 and c_2 with their least common subsumer, etc.). Such similarity functions do not take into account the distribution of the instances in the set of cases, in comparison with our cost function which represents a generalization effort. Moreover, the cost function is independent of the level of structuration of the ontology. So, having many intermediate concepts (like for example `CitrusFruitJuice` between `OrangeJuice` and `FruitJuice`) do not impact the case retrieval. This point is crucial for improving case retrieval with a better structured ontology [8].

myCBR [9] is a Java open source similarity-based tool. Cases, described by attributes with different weights, may be created thanks to a graphical user interface or through a file import (e.g., CSV file). A "Linked Open Data Connector" provides an access to open data sources for building taxonomies. Many similarity functions are provided and may be combined to rank cases according to their similarity with the query (during the retrieval step). The adaptation consists in replacing some case features with others, using adaptation rules retrieved from the myCBR knowledge models (e.g. similarity tables or taxonomies). So, the myCBR retrieving/adapting procedure is rather similar to the jCOLIBRI one.

Like myCBR and jCOLIBRI, TUUURBINE is a generic CBR engine. The originality of TUUURBINE is the implementation of a CBR based on RDFS and the exploitation of the semantics of the RDF and RDFS models. The whole knowledge (cases, domain knowledge, costs, adaptation rules) is described using RDF and RDFS and is managed outside TUUURBINE using a triple store and SPARQL, two standard tools of the Semantic Web. The edition of the knowledge is also facilitated with the use of a Semantic Wiki, another Semantic Web tool.

7 Conclusion

This paper has presented TUUURBINE, a generic CBR engine based on RDFS and using Semantic Web technologies. The main principles for representation of the four knowledge containers as well as the reasoning processes have been detailed. Information about technical use and configuration of TUUURBINE is available at `http://tuuurbine.loria.fr/`. TUUURBINE is used in the new version of the TAAABLE system, a CBR system adapting cooking recipes, which will participate in the 2014's Computer Cooking Contest.

The current version of the TUUURBINE case-based inference engine using query generalizations is based on the `subc` property (generalization of a class by a super-class). Other constructs of RDFS could be used as well in the TUUURBINE reasoning process: the subproperty relation (`subp`), domains and ranges of properties, literals (e.g., numerical values). This constitutes a future direction of research. It is planned to address first the use of `subp`. For instance, if $\langle$`mainIngredient subp ingredient`$\rangle \in$ `DK` (if x is a main ingredient of y then

x is an ingredient of y), the query $\mathtt{Q} = \mathtt{mainIngredient:TomatoJuice}$ can be generalized into $\Gamma(\mathtt{Q}) = \mathtt{ingredient:TomatoJuice}$.

Another direction of work related to RDFS expressiveness consists in managing "deeper" cases: in the current implementation, only values directly related to s are taken into account, but such values could be related to other values participating to the case representation. Following this direction would make TUUURBINE evolve from an attribute-value representation to something similar to an object-based representation (like the ones of jCOLIBRI and myCBR).

A third direction of work is the integration of TUUURBINE with other generic CBR systems such as jCOLIBRI and myCBR (cf. Section 6) or Revisor [10].

References

1. A. Cordier, V. Dufour-Lussier, J. Lieber, E. Nauer, F. Badra, J. Cojan, E. Gaillard, L. Infante-Blanco, P. Molli, A. Napoli, and H. Skaf-Molli. Taaable: a Case-Based System for personalized Cooking. In Stefania Montani and Lakhmi C. Jain, editors, *Successful Case-based Reasoning Applications-2*, volume 494 of *Studies in Computational Intelligence*, pages 121–162. Springer, January 2014.
2. A. Cordier, J. Lieber, P. Molli, E. Nauer, H. Skaf-Molli, and Y. Toussaint. WIKITAAABLE: A semantic wiki as a blackboard for a textual case-based reasoning system. In *SemWiki 2009 - 4rd Semantic Wiki Workshop at the 6th European Semantic Web Conference - ESWC 2009*, Heraklion, Grèce, May 2009.
3. F. Badra, R. Bendaoud, R. Bentebibel, P.-A. Champin, J. Cojan, A. Cordier, S. Després, S. Jean-Daubias, J. Lieber, T. Meilender, A. Mille, E Nauer, A. Napoli, and Y. Toussaint. TAAABLE: Text Mining, Ontology Engineering, and Hierarchical Classification for Textual Case-Based Cooking. In Martin Schaaf, editor, *9th European Conference on Case-Based Reasoning - ECCBR 2008, Workshop Proceedings*, pages 219–228, Trier, Allemagne, 2008.
4. W. Shakespeare. *Romeo and Juliet*. 1597.
5. M. M. Richter. Introduction. In M. Lenz, B. Bartsch-Spörl, H.-D. Burkhard, and S. Wess, editors, *Case-Based Reasoning Technologies. From Foundations to Applications*, LNAI 1400, chapter 1, pages 1–15. Springer, 1998.
6. K. Maximini, R. Maximini, and R. Bergmann. An investigation of generalized cases. In K. D. Ashley and D. Bridge, editors, *Proceedings of the 5th International Conference on Case Base Reasoning (ICCBR'03)*, volume 2689 of *LNAI*, pages 261–275, Trondheim, Norway, June 2003. Springer.
7. Juan A. Recio-García, Pedro A. González-Calero, and Belén Díaz-Agudo. jcolibri2: A framework for building case-based reasoning systems. *Science of Computer Programming*, (79):126–145–, 2014.
8. V. Dufour-Lussier, J. Lieber, E. Nauer, and Y. Toussaint. Improving case retrieval by enrichment of the domain ontology. In *19th International Conference on Case Based Reasoning - ICCBR'2011*, London, 2011.
9. K. Bach and K. Althoff. Developing case-based reasoning applications using mycbr 3. In Belén Díaz-Agudo and Ian Watson, editors, *ICCBR*, volume 7466 of *LNAI*, pages 17–31. Springer, 2012.
10. J. Cojan and J. Lieber. Applying belief revision to case-based reasoning. In Henri Prade and Gilles Richard, editors, *Computational Approaches to Analogical Reasoning: Current Trends*, volume 548 of *Studies in Computational Intelligence*. Springer, 2014.